Práctica 3: Resolución de problemas de sincronización mediante semáforos

Programación de Sistemas Concurrentes y Distribuidos

Dpto. de Informática e Ingeniería de Sistemas,
Grado de Ingeniería Informática
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

1. Objetivos

En esta práctica se estudiará la resolución de problemas de sincronización mediante semáforos. En concreto, los objetivos de esta práctica son:

- comprender y profundizar en la sincronización de procesos,
- resolver problemas de sincronización de procesos utilizando semáforos,
- y profundizar en el modelo de concurrencia de C++.

2. Trabajo previo a la sesión en el laboratorio

Antes de la correspondiente sesión en el laboratorio, cada pareja de estudiantes deberá leer el enunciado, analizar los problemas que en él se proponen y realizar un diseño previo de las soluciones sobre las que va a trabajar. *Los resultados de su trabajo de análisis y diseño los tendrá que expresar en un documento que entregará y presentará a los profesores antes del inicio de la sesión.* El documento debe contener como mínimo el nombre completo y el NIP de los dos estudiantes y, para cada ejercicio,

- un diseño de alto nivel de la solución que incluya la descripción de los datos compartidos, enumeración de los procesos que los comparten e identificación de los semáforos necesarios para sincronizar la actividad de estos procesos.
- un esbozo de alto nivel del código de los procesos indicando las zonas que están afectadas por la sincronización.

El documento deberá llamarse `informe_P3_NIP1_NIP2.pdf` (donde `NIP1` **es el NIP menor** y `NIP2` **es el NIP mayor** de la pareja), y deberá entregarse antes del comienzo de la sesión de prácticas utilizando el comando `someter` en la máquina `hendrix.cps.unizar.es`

```
someter progc_20 informe_P3_NIP1_NIP2.pdf
```

Su entrega es un pre-requisito para la realización y evaluación de la práctica.

3. Generación de ficheros de log

En muchas situaciones es interesante almacenar información sobre la historia de ejecución, con el fin de poder analizar propiedades comportamentales, detectar problemas e inconsistencias, etc. En el caso de un programa secuencial es sencillo: se escriben eventos en un fichero conforme van sucediendo. En el caso de un programa concurrente, por cuestiones de entrelazados no es tan sencillo, ya que puede que los eventos no se guarden en el orden en que ocurrieron. Para ello suministramos la librería `logger_V2`. La generación del log se puede usar solo durante la etapa de desarrollo, para poner a punto el programa, pero también se puede dejar para la etapa de explotación, pues podría ser útil para detectar posibles causas de problemas. En cualquier caso, que el programa genere un fichero de log o no se puede activar/desactivar en la compilación del mismo.

Una forma de compilar programas con y sin la opción de generar el log es la que se muestra en el ejemplo `pruebaSemaforos.cpp`, mediante la compilación condicional definida por `#ifdef LOGGING_MODE ... #else ... #endif` en el main, junto con `-D LOGGING_MODE` en la invocación al compilador en el Makefile correspondiente. Así, si en la invocación al compilador en el makefile aparece la opción `-D LOGGING_MODE`, entonces se define la variable interna `LOGGING_MODE`. Si no aparece, no está definida. Por otro lado, las líneas 20 a 27 del fichero `pruebaSemaforos.cpp` analizan si dicha variable está definida. Si no lo está, se ejecutará la línea 26, con lo que `ADD_EVENT(e)` será la instrucción vacía. En este caso, la línea 95 es lo mismo que una línea vacía. En caso contrario se ejecutará las líneas 21 (incluye las definiciones del logger), la 22 (generando el fichero de log `_log_.log`) con un buffer de capacidad 1024 (lo eventos se van guardando en una estructura de datos intermedia y son pasados al fichero cuando se hayan generado 1024 eventos o cuando se acabe el programa), y haciendo que la instrucción `ADD_EVENT` se defina como la línea 24 establece. Ahora la línea 95 ya no será vacía, sino que será, exactamente, {`_logger.addMessage(e);`}. Esta instrucción almacena un evento en el fichero de log. El evento se forma a partir del parámetro `e` (un string) junto con información del sistema (identificador del thread que lo ha generado, el timestamp del momento en que ha sucedido, ...). El contenido de `e` dependerá del programa concreto, que es lo que le dará el significado.

4. Semáforos en C++

C++ no tiene datos de tipo semáforo con la semántica vista en clase. Por este motivo se suministra la clase `Semaphore_V4`, correspondiente a los ficheros `Semaphore_V4.hpp` y

`Semaphore_V4.cpp`. Se proporciona también un ejemplo de uso `pruebaSemaforos.cpp`.

La clase implementada tiene un único constructor `Semaphore(const int n, const string info = '')`. El primer parámetro (obligatorio) corresponde al número de permisos que se le asocia inicialmente, equivalente al usado en la notación algorítmica utilizada en clases de teoría. El segundo parámetro es opcional, y se podrá usar, cuando así lo consideremos oportuno, para cuestiones de *logging*: para almacenar y analizar información sobre la historia de ejecución del programa. Ese parámetro lo usa la clase semáforo para generar automáticamente información para el log. Dado que en esta práctica no la vamos a usar, inicialmente, no incidimos en ello. Queda pendiente para más adelante.

La especificación de la clase semáforo ofrece dos versiones de `wait` y dos de `signal`. La diferencia entre ellas radica en el número de permisos que están involucrados en la acción. La versión sin parámetros (`sem.signal(), sem.wait()`) se corresponde con la explicada en clase (el valor del semáforo se incrementa/decrementa en una unidad respetando las reglas semánticas de la primitiva de sincronización). La semántica de la segunda versión (`sem.signal(3), sem.wait(2)`, por ejemplo), es una generalización de la anterior, que se explica en los comentarios del fichero `Semaphore_V4.hpp`.

5. Ejercicio a desarrollar

Hay un tren turístico, para visitar parajes protegidos, compuesto por dos vagones: uno, más moderno, con capacidad para 4 viajeros y otro, más antiguo, para 2 viajeros. Cuando un usuario llega al arcén, si las puertas de entrada están abiertas y hay hueco en el primer vagón, ocupa uno de sus asientos. Si está lleno, pero hay hueco en el segundo, se sienta allí. Si al llegar ambos están llenos o las puertas de entrada están cerradas, tendrá que esperar al siguiente viaje. Cada vez que un viajero sube a un vagón pasa su billete por un lector, que actualiza los contadores del sistema del número de viajeros en cada vagón.

Por su parte, el conductor espera a que el tren esté lleno (observando los valores de los contadores de ocupación del sistema), momento en el que cierra las puertas de entrada y parte de viaje. Asumimos que el viaje dura un tiempo aleatorio de entre 10 y 15 milisegundos (que simulan una duración entre 10 y 15 minutos).

Una vez termina el viaje, el conductor abre las puertas de salida y espera a que el sistema detecte que cada uno de los viajeros ha descendido del vagón. En este momento abre las puertas de entrada, y se inicia un nuevo viaje.

Se trata de implementar un programa que simule el comportamiento del sistema, asumiendo que hay 48 viajeros. Cada viajero, lo mismo que el conductor, se implementará mediante un proceso.

El desarrollo de la práctica se llevará a cabo como sigue:

- Plantéese un primer diseño de la solución con la notación algorítmica presentada en clase, utilizando instrucciones del tipo *await*, y acorde con el esquema mostrado en el Anexo-I.

- Llévese a cabo una primera implementación aplicando la técnica del paso de testigo. Es posible utilizar, para esta versión, variables globales, de manera análoga al diseño planteado (puntuación hasta 8.5/10.0).
- De manera opcional, se puede llevar a cabo la implementación sin el uso de variables globales, parametrizando de manera adecuada los procedimientos que ejecutan los threads. Por simplicidad, dejaríamos como única variable global el `logger`.

El Anexo-I muestra un esbozo de código para los procesos involucrados, así como las variables que se pueden manejar.

6. Entrega de la práctica

Cuando la práctica se finalice, los dos componentes de la pareja deben entregar, cada uno desde su cuenta, el mismo fichero comprimido `practica_3_NIP1_NIP2.zip` (donde `NIP1` **es el NIP menor** y `NIP2` **es el NIP mayor** de la pareja) con el siguiente contenido:

1. El fichero `disegno.pdf` con el diseño en base a la instrucción `await` que se pide
2. El fichero `practica_3.cpp` con el main de programa
3. El directorio `librerias` que se suministra (que, a su vez, contiene las librerías de semáforos y para generar ficheros de log)
4. El fichero `Makefile_p3` que compila el fuente, generando el ejecutable `practica_3`
5. Todos los demás ficheros requeridos para que la ejecución de `make -f Makefile_p3` genere el ejecutable pedido

Generación del fichero `.zip` a entregar

Con el objetivo de homogeneizar los contenidos del fichero `.zip` vamos a proceder como sigue:

1. Creamos un directorio `practica_3_NIP1_NIP2` que contenga los ficheros que hay que entregar. Es importante tener presente que **se ha de hacer exactamente de esta manera**.
2. Con el botón derecho del ratón sobre la carpeta seleccionamos la opción "Compress..." y le damos en nombre requerido, `practica_3_NIP1_NIP2.zip`
3. Alternativamente lo podemos hacer desde la terminal como sigue. Una vez creado el directorio `practica_3_NIP1_NIP2` con los ficheros pedidos ejecutamos lo siguiente desde la terminal:

```
zip -r practica_3_NIP1_NIP2.zip practica_3_NIP1_NIP2
```

Con el fin de comprobar que el `zip` contiene todos los ficheros que debe, y organizados adecuadamente, podéis ejecutar el script `pract_3_entrega_correcta.bash`. Leed la cabecera del fichero, que explica cómo utilizarlo.

Entrega del fichero en `hendrix`

Para la entrega del fichero `.zip` se utilizará el comando `someter` en la máquina `hendrix.cps.unizar.es`

```
someter progc_20 practica_3_NIP1_NIP2.zip
```

Fechas de entrega de la práctica

Los alumnos pertenecientes a grupos de prácticas cuya primera sesión de prácticas se celebra el día 4 de noviembre del 2020 deberán someter la práctica no más tarde del día 13 de noviembre del 2020 a las 23:59. Los alumnos pertenecientes a grupos de prácticas cuya primera sesión de prácticas se celebra el día 11 de noviembre del 2020 deberán someter la práctica no más tarde del día 20 de noviembre del 2020 a las 23:59.

Hay que asegurarse de que la práctica funciona correctamente en los ordenadores del laboratorio (hay que vigilar aspectos como los permisos de ejecución, juego de caracteres utilizado en los ficheros, etc.). También es importante someter código limpio (donde se ha evitado introducir mensajes de depuración que no proporcionan información al usuario). El tratamiento de errores debe ser adecuado, de forma que si se producen debería informarse al usuario del tipo de error producido. Además se considerarán otros aspectos importantes como calidad del diseño del programa, adecuada documentación de los fuentes, correcto formateado de los fuentes, etc.

Para el adecuado formateado de los fuentes, es conveniente seguir unas pautas. Hay varias, y es posible que podáis configurar el entorno de desarrollo para cualquiera de ellas. Una posible, sencilla de seguir, es la "Google C++ Style Guide", que se puede encontrar en

`https://google.github.io/styleguide/cppguide.html`

Alternativamente, cualquiera que uséis en otras asignaturas de programación.

Anexo-I

Esbozo del código de los procesos

```
1 const int N := 48 //num de usuarios
2 const int N_VIAJES := 8 //num total de viajes
3
4 const int N_W1 = 4 //capacidad vagón 1
5 const int N_W2 = 2 //capacidad vagón 2
6
7 int w1 := 0 //personas dentro de vagón 1
8 int w2 := 0 //personas dentro de vagón 2
```

```
9
10 int hanBajado := 0 //+1 cada vez que un usuario se baja de su vagón
11 bool puertasEntradaAbiertas := true //estado de las puertas
12 bool puertasSalidaAbiertas := false
13
14 Process usuario(i: 1..N):
15   //esperar a que las puertas de entrada estén abiertas
16   //y haya hueco. Entonces, entrar y generar evento
17   //ADD_EVENT("MONTA,"+to_string(i)+","+to_string(w1)+","+to_string(w2));
18
19   //esperar a que las puertas de salida estén abiertas,
20   //y entonces salir y generar evento
21   //ADD_EVENT("DESMONTA,"+to_string(i)+","+to_string(w1)+","+to_string(w2));
22 end
23
24 Process conductor:
25   for i:=1..N_VIAJES
26     //esperar a que los vagones se llenen; entonces cerrar las
27     //puertas de entrada y generar evento
28     //ADD_EVENT("INICIO_VIAJE,1000,"+to_string(w1)+","+to_string(w2));
29
30     //viajar. Tiempo aleatorio
31
32     //abrir las puertas de salida y generar evento
33     //ADD_EVENT("FIN_VIAJE,1000,"+to_string(w1)+","+to_string(w2));
34
35     //esperar a que todos bajen; entonces actualizar
36     //el estado de las puertas y generar evento
37     //ADD_EVENT("TODOS_HAN_BAJADO,1000,"+to_string(w1)+","+to_string(w2));
38   end
39 end
```

Por otra parte, el `main` generará los eventos siguientes justo al principio y al final, respectivamente,

```
1 ADD_EVENT("BEGIN_MAIN,0,"+to_string(w1)+","+to_string(w2));
2 ...
3 ADD_EVENT("END_MAIN,0,"+to_string(w1)+","+to_string(w2));
```

Anexo-II

Un posible comienzo de fichero de log para el programa solicitado.

```
1 ID,event,procId,val1,val2,ts,threadID,ticket
2 id_140599180977984,BEGIN_MAIN,0,0,0,1603708760771402430,140599180977984,1
3 id_140599180977984,MONTA,1,1,0,1603708760772024733,140599164188416,2
4 id_140599180977984,MONTA,5,2,0,1603708760772431226,140599130617600,3
5 id_140599180977984,MONTA,7,3,0,1603708760772756184,140599042766592,4
6 id_140599180977984,MONTA,3,4,0,1603708760773106867,140599147403008,5
```

```
id_140599180977984,MONTA,18,4,1,1603708760773631331,140598874978048,6
id_140599180977984,MONTA,9,4,2,1603708760773941971,140599025981184,7
id_140599180977984,INICIO_VIAJE,1000,4,2,1603708760774522355,140599180973824,8
id_140599180977984,FIN_VIAJE,1000,4,2,1603708760781596468,140599180973824,9
id_140599180977984,DESMONTA,18,4,1,1603708760784874179,140598874978048,10
...
```